Spring 5-1-1991

Software Issues at the User Interface ; CU-CS-527-91

Oliver A. McBryan
University of Colorado Boulder

Software Issues at the User Interface

Oliver A. McBryan

CU-CS-527-91 May 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-3898
(303) 492-2844 Fax
mcbryan@cs.colorado.edu

SOFTWARE ISSUES AT THE USER INTERFACE[†]

Oliver A. McBryan*

Department of Computer Science
University of Colorado
Boulder, CO 80309

ABSTRACT

We review software issues that are critical to the successful integration of parallel computers into mainstream scientific computing. Clearly a compiler is the most important software tool available to a user on most systems. We discuss compilers from the point of view of communication compilation - their ability to generate efficient communication code automatically. We illustrate with two examples of distributed memory computers where almost all communication is handled by the compiler rather than by explicit calls to communication libraries.

Closely related to compilation is the need for high quality debuggers. While single node debuggers are important, parallel machines have their own specialized debugging needs related to the complexity of interprocess communication and synchronization. We describe a powerful simulation tool we have developed for such systems and which has proved essential in porting large applications to distributed memory systems.

Other important software tools include high level languages, libraries and visualization software. We discuss aspects of these systems briefly. Ultimately however, general purpose supercomputing environments are likely to include more than a single computer system. Parallel computers are often highly specialized, and rarely provide all of the facilities required by a complete application. Over the coming decade we will see the development of heterogeneous environments connecting diverse supercomputers (scalar, vector and parallel) along with high end graphics, disk farms and networking hubs. The real user interface challenge will then be to provide a unified picture of such systems to potential users.

---

† Presented at Frontiers of Supercomputing II: A National Reassesessment, Los Alamos National Laboratory, August 1990.

* Research supported by the Air Force Office of Scientific Research, under grant AFOSR-89-0422

1. INTRODUCTION

This paper will survey a selection of issues dealing with software and the user interface, always in the context of parallel computing.

The most obvious feature in looking back over the last five to ten years of parallel computing is that it has been clearly demonstrated that parallel machines can be built. This was a significant issue eight or ten years ago, particularly in the light of some early experiences. There was a feeling that the systems would be too unreliable, with mean time to failure measured in minutes, and intractable cooling and packaging issues. There are now several examples of real highly parallel systems such as the Connection Machine CM-2 and the iPSC/860, and these are certainly recognized as serious computers and among the leading supercomputers at the present time.

It has also been shown that model problems can be solved efficiently on these systems. For example, linear algebra problems, partial differential equation solvers, and simple applications such as QCD have been modeled effectively. In a few cases, more complex models such as oil reservoir simulations and weather models have been parallelized.

While hardware progress has been dramatic, system software progress has been painfully slow, with a few isolated exceptions which I will highlight in the following section. Another area in which there has been almost no progress is in demonstrating that parallel computers can support general purpose application environments. Most of what we will present here will be motivated to some extent by these failures.

We could begin by asking what a user would like from a general purpose supercomputer application environment. In simplest terms, he would like to see a computer consisting of a processor that is as powerful as he wants, a data memory as large as his needs, and massive connectivity with as much bandwidth (internal and external) as desired. Basically he would like the supercomputer to look like his desk top workstation. This also corresponds to the way he would like to program the system.

It is well known that there are various physical limitations such as the finite speed of light and cooling issues that prevent us from designing such a system. As a way around, one has been led to parallelism which replicates processors, memories and data paths in order to achieve comparable power and throughput. To complexify the situation further there are many different ways to actually effect these different connections, as well as several distinctly different choices for control of such systems. Thus we end up with the current complexity of literally dozens of different topologies and connection strategies.

Yet the user wants to think of this whole system as a monolithic processor if he possibly can. We will focus on several software areas where the user obviously interacts with the system, and discuss ways in which the software can help with this issue of focusing on the machine as a single entity.

At the lowest level we will discuss compilers, but not from the point of view of producing good machine code for the individual processors, but rather from the higher level aspect of how the compiler can help with the unifying aspect for parallel machines.

Then there is the all-important debugging, trace and simulation phases. People actually spend most of their time developing programs rather than running them, and if this phase is not efficient for the user, the system is likely to ultimately be a failure.

We will briefly discuss several related developments: higher-level languages, language extensions and libraries. Portable parallel libraries provide a key way in which the user's interaction with systems can be simplified. Graphics and the visualization pipeline is of course a critical area about which we will make several comments. For each of these topics we will refer to other papers in the volume for more coverage.

Finally, we will discuss the development of software for heterogeneous environments. which is in many ways the most important software issue. No one parallel machine is going to turn out to provide the general-purpose computer that the user would really like to have. There will be classes of parallel machines, each well suited to a range of problems, but unable to provide a solution environment for the full range of applications. Heterogeneous systems will allow complex algorithms to avail of appropriate and optimal resources as needed. So ultimately we will be building heterogeneous environments, and the software for those systems is perhaps the greatest challenge in user interface design in the near future.

2. COMPILERS AND COMMUNICATION

There are three roles that compilers play in the context of parallel machines. First of all, they provide a mechanism for generating good scalar and vector node code. Since that topic is covered adequately in other papers in this volume, we will not focus on it here. Rather we will focus on the fact that the compiler can help the user by taking advantage of opportunities for automatic parallelization, and particularly important in the context of distributed machines, there is the possibility for compilers to help the user with some of the communication activities.

The current compilers do a very good job in the area of scalar/vector node code generation, although some node architectures (e.g. i860) are quite a challenge to compiler writers. Some of the compilers also make a reasonable effort in the area of parallelization, at least in cases where data dependencies are obvious. However, there is very little to point to in the third area - of compilers helping on distributed machines. The picture here is not completely bleak, so we will refer to two examples that really stand out, namely Thinking Machines Corporation's Connection Machine CM-2 and Myrias Research's SPS-2 computers. In both of these systems the compilers and the associated run-time system really help enormously with instantiation and optimization of communication.

2.1. Myrias SPS-2: Virtual Memory on a Distributed System

The Myrias SPS-2 system was introduced in Gary Montry's presentation. It is a typical distributed-memory machine, based on local nodes (Motorola 68020) with some memory associated, and connected by busses organized in a three-level hierarchy. The SPS-2 has the remarkable feature that it supports a virtual shared memory, and that feature is what we want to focus on here. For further details on the SPS-2 we refer to our paper[1].

On the system side, virtual shared memory is implemented by the Fortran compiler and by the run-time system. The result is to present a uniform 32-bit address space to any program, independent of the number of processors being used. From the user's point of view, he can write a standard Fortran F77 program, compile it on the machine, and run it as is, *without any code modification.* The program will execute instructions on only one processor, (assuming it is written in standard Fortran), but it may use the memory from many processors. Thus even without any parallelization, programs automatically use the multiple memories of the system, through the virtual memory. For example, a user could take a large CRAY application, with a data requirement of Gbytes, and would have it running immediately on the SPS-2 despite the fact that each node processor has only 8 Mbytes of memory.

With the sequential program now running on the SPS-2, the next step is to enhance performance by exploiting parallelism at the loop level. In order to parallelize the program, one seeks out loops where the internal code in the loop involves no data dependencies between iterations. Replacing DO by PARDO in such loops parallelizes them. This provides the mechanism to use not only the multiple memories, but also the multiple processors.

Developing parallel programs then becomes a two-step refinement: first of all use multiple memories by just compiling the program, and secondly adding PARDOs to achieve instruction parallelism.

As discussed in the following section, the virtual memory support appears to reduce SPS-2 performance by about 40-50%. A lot of people would regard a 50-percent efficiency loss as too severe. But we would argue that if one looks at the software advantages over long term projects of being able to implement shared-memory code on a distributed-memory system, those 50-percent losses are certainly affordable. However one should note that the SPS-2 is not a very powerful supercomputer as the individual nodes are 150 Kflops 68020 processors. It remains to be demonstrated that virtual memory can run on more powerful distributed systems with reasonable efficiency.

One other point that should be made is that we are not talking about virtual shared memory on a shared-memory system. The SPS-2 computer is a true distributed-memory system. Consequently one cannot expect that just any shared memory program will run efficiently. In order to run efficiently, a program should be well suited to distributed systems to begin with. For example, grid-based programs that do local access of data will run well on such a system. Thus while you can *run* any program on these systems without modification, you can only expect good performance from programs that access data in the right way.

The real benefit of the shared memory to the user is that he doesn't have to consider the layout of data. Data flows naturally to wherever it is needed, and that is really the key advantage to the user of such systems. In fact for dynamic algorithms, extremely complex load balancing schemes have to be devised in order to accomplish what the SPS-2 system does routinely. Clearly such software belongs in the operating system, and not explicitly in every users programs.

2.1.1. Myrias SPS-2: A Concrete Example

In this section we study simple relaxation processes for 2D Poisson equations in order to illustrate the nature of a Myrias program.. These are typical of processes occurring in many applications codes involving either elliptic PDE solution or time evolution equations. The most direct applicability of these measurements is to performance of standard "fast solvers" for the Poisson equation. The code kernels we will describe are essentially those used in relaxation, multigrid and conjugate gradient solution of the Poisson equation. Because the Poisson equation has constant coefficients, the ratio of computational work per grid point to memory traffic is severe, and it is fair to say that while typical, these are very hard PDE to solve efficiently on a distributed memory system.

The relaxation process has the form:

$$v(j,i) = s\,u(j,i) + r(u(j+1,i) + u(j-1,i) + u(j,i+1) + u(j,i-1)),$$

Here the arrays are of dimensions $n_1 \times n_2$ and s, r are specified scalars, often 4 and 1 respectively. The equation above is to be applied at each point of the *interior* of a 2D rectangular grid, which we will denote generically as G. Were the equations to be applied at the boundary of G, then they would index undefined points on the right hand side. This choice of relaxation scheme corresponds to imposition of Dirichlet boundary conditions in a PDE solver. The process generates a new solution v from a previous solution u. The process is typified by the need to access a number of nearby points. At the point i,j it requires the values of u at the four nearest neighbors.

We implement the above algorithm serially by enclosing the expression in a nested set of DO loops, one for each grid direction.

```
      do 10 j = 2,n1-1
      do 10 i = 2,n2-1
          v(j,i) = s*u(j,i) + r(u(j,i-1) + u(j,i+1) + u(j-1,i) + u(j+1,i))
10    continue
```

To parallelize this code using T parallel tasks, we would like to replace each DO with a PARDO, but this in general generates too many tasks - a number equal to the grid size. Instead we will decompose the grid G into T contiguous rectangular subgrids and each of T tasks will be assigned to process a different subgrid.

The partitioning scheme used is simple. Let $T=T_1T_2$ be a factorization of T. Then we divide the index interval $[2,n_1-1]$ into T_1 essentially equal pieces and similarly we divide $[2,n_2-1]$ into T_2 pieces. The tensor product of the interval decompositions defines the 2D subgrid decomposition.

In case T_1 does not divide n_1-2 evenly, we can write:

$$n_1-2 = h_1T_1 + r_1,\ 0\le r_1<T_1.$$

We then make the first r_1 intervals of length h_1+1 and the remaining T_1-r_1 intervals of length h_1, and similarly in the other dimension(s). This is conveniently done with a procedure

$$decompose\,(a,b,t,istart,iend)$$

which decomposes an interval $[a,b]$ into t near-equal length subintervals as above, and which initializes arrays $istart(t)$, $iend(t)$ with the start and end indices of each subinterval.

Thus the complete code to parallelize the above loop takes the form:

```
      decompose(2,n1-1,t1,istart1,iend1)
      decompose(2,n2-1,t2,istart2,iend2)
      pardo 10 q1=1,t1
      pardo 10 q2=1,t2
          do 10 i= istart1(q1),iend1(q1)
          do 10 j= istart2(q2),iend2(q2)
              v(j,i) = s*u(j,i) + r(u(j,i-1) + u(j,i+1) + u(j-1,i) + u(j+1,i))
10    continue
```

The work involved in getting the serial code to run on the Myrias using multiple processors involved just one very simple code modification. The DO loop over the grid points is replaced by, first of all, a DO loop over processors, or more correctly tasks. Each task computes the limits within the large array that it has to work on by some trivial computation. And then the task goes ahead and works on that particular limit. However the data arrays for the problem were never explicitly decomposed by the user as would be needed on any other distributed memory MIMD machine.

This looks exactly like the kind of code you would write on a shared-memory system. Yet the SPS-2 is truly a distributed-memory system. It really is similar to an Intel Hypercube from the logical point of view. It is a fully distributed system, and yet you can write code like this that has no communication primitives. That is a key advance in the user interface of distributed-memory machines, and we will certainly see more of this approach in the future.

2.1.2. MYRIAS SPS-2: Efficiency of Virtual Memory

We have made numerous measurements on the SPS-2 that attempt to quantify the cost of using the virtual shared memory in a sensible way[1]. One of the simplest tests is a *saxpy* operation (adding a scalar times a vector to a vector):

$$y_i = y_i + a\, x_i \quad .$$

We look at the change in performance as the vector is distributed over multiple processors, while performing all computations using only one processor. Thus we take the same vector, but allow the system to spread it over varying numbers of processors, and then compute the *saxpy* using just one processor. We define the performance with one processor in the domain as efficiency 1. As soon as one goes to two or more processors there is a dramatic drop in efficiency to about 60%, and performance stays at that level more or less independent of the numbers of processors in the domain. That then measures the overhead for the virtual shared memory.

A lot of people would regard a 50-percent efficiency loss as too severe. We would argue that if you look at the software advantages over long-term projects of being able to implement shared-memory code on a distributed-memory system, then 50-percent losses are certainly affordable.

Another aspect of efficiency related to data access patterns may be seen in the relaxation example presented in the previous section. The above procedure provides many different parallelizations of a given problem, one for each possible factorization of the number of tasks T. At one extreme are decompositions by rows (case T_1=1), and at the other extreme are decompositions by columns (T_2=1), with intermediate values representing decompositions by subrectangles. Performance is strongly influenced by which of these choices is made. We have in all cases found that decomposition by columns gives maximum performance. This is not a priori obvious, as in fact area-perimeter considerations would suggest that virtual memory communication would be minimized with a decomposition where $T_1 = T_2$. Two competing effects are at work: the communication bandwidth requirements are determined by the perimeter of subgrids, whereas communication overhead costs (including memory merging on task completion) are determined additionally by a factor proportional to the total number of data requests. The latter quantity is minimized by a column division. Row-division is unfavorable because of the Fortran rules for data storage.

It is instructive to study the variation in performance for a given task number T as the task decomposition varies - we refer to this as "varying the subgrid aspect ratio", although in fact it is the *task subgrid* aspect ratio. We present sample results for 2D relaxations in Table 1. The efficiency measures the deviation from the optimal case. Not all aspect ratios would in fact run. For heavily row-oriented ratios (e.g. T_1=1, T_2=T) the system runs out of virtual memory and kills the program unless the grid size is quite small.

Table 1: 2D Effect of Subgrid Aspect Ratio

Grid	D	T1	T2	Mflops	Efficiency
512×512	64	1	64	.036	.022
512×512	64	2	32	.076	.047
512×512	64	4	16	.217	.134
512×512	64	8	8	.502	.310
512×512	64	16	4	.946	.584
512×512	64	32	2	1.336	.825
512×512	64	64	1	1.619	1.000

2.2. The Connection Machine CM-2: Overlapping Communication with Computation

The Connection Machine CM-2 provides another good example of how a powerful compiler can provide a highly effective user interface and free the user from most communication issues. The Connection Machine is a distributed memory (hypercube) SIMD computer which in principle might have been programmed using standard message passing procedures. For a more detailed description of the CM-2 see[2] In fact the assembly language of the system supports such point to point communication and broadcasting. However Connection Machine high level software environments provide basically a virtual shared memory view of the system. Each of the three high level supported languages: CM Fortran, C* and *Lisp, makes the system look to the user as if he is using an extremely powerful uniprocessor with an enormous extended memory. These languages support parallel extensions of the usual arithmetic operations found in the base language, which allows SIMD parallelism to be specified in a very natural and simple fashion. Indeed CM-2 programs in Fortran or C* are typically substantially shorter than their serial equivalents from workstations or CRAY's, because DO loops are replaced by parallel expressions.

However in this discussion I would like to emphasize that very significant communication optimization is handled by the software. This is best illustrated by showing the nature of the optimizations involved in code generation for the same generic relaxation type operation discussed in the previous section. We will see that without communication optimization the algorithm runs at around 800 megaflops, which increases to 3.8 gigaflops when compiler optimizations are used to overlap computation and communication..

For the simple case of a Poisson type equation, the fundamental operation $v = Au$ takes the form (with r and s scalars):

$$v_{i,j} = su_{i,j} + r(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) \ .$$

The corresponding CM-2 parallel Fortran takes the form:

```
v = s*u + r*(cshift(u,1,1) + cshift(u,1,-1) + cshift(u,2,1) + cshift(u,2,-1))  .
```

Here *cshift* (u,d,l) is a standard Fortran 90 array operator which returns the values of a multi-dimensional array u at points a distance l away in dimension direction d.

The equivalent *LISP version of a function *applya* for $v = Au$ is:

```
(defun *applya (u v)

     (*set v  (-!! (*!! (!! s) u)
                   (*!! (!! r) (+!! (news!! u -1 0) (news!! u 1 0)
                                    (news!! u 0 -1) (news!! u 0 1)
)))))
```

*Lisp uses !! to denote parallel objects or operations, and as a special case, !! s is a parallel replication of a scalar s. Here *(news!! u dx dy)* returns in each processor the value of parallel variable u at the processor dx processors away horizontally and dy away vertically. Thus *cshift* $(i+1,j)$ in Fortran would be replaced by *(news!! u 1 1)* in *Lisp.

The *Lisp source shown was essentially the code used on the CM-1 and CM-2 implementation described in[3]. When first implemented on the CM-2 it yielded a solution rate of only .5 Gflops. Many different optimization steps were required to raise this performance to 3.8 Gflops over a one year period. Probably the most important series of optimizations turned out to be those involving the overlap of communication with computation. Working with compiler and microcode developers at Thinking Machines Corporation, we determined the importance of such operations, added them to the microcode and finally improved the compiler to the point where it automatically generated such microcode calls when presented with the source above.

We will illustrate the nature of the optimizations by discussing the assembly language (called **PARIS** an acronym for **PAR**allel **I**nstruction **S**et) code generated by the optimized compiler for the above code fragment. The PARIS code generated by the optimizing *Lisp compiler under version 4.3 of the CM-2 system is shown in the display below. Here the code has expanded to generate various low level instructions, with fairly recognizable functionality, including several that overlap computation and communication such as:

$$cmi:get-from-east-with-f-add-always.$$

which combines a communication (getting data from the east) with a floating-point operation (addition).

Optimized PARIS Code for Relaxation:

```
(defun *applya (u v)
    (let* ((slc::stack-index *stack-index*)
         (-!!-index-2 (+ slc::stack-index 32))
         (pvar-location-u-11 (pvar-location u))
         (pvar-location-v-12 (pvar-location v)))

      (cm:get-from-west-always -!!-index-2 pvar-location-u-11 32)
      (cm:get-from-east-always *!!-constant-index4 pvar-location-u-11 32)
      (cmi::f+always -!!-index-2 *!!-constant-index4 23 8)
      (cmi::get-from-east-with-f-add-always -!!-index-2 pvar-location-u-11 23 8)

      (cmi::f-multiply-constant-3-always pvar-location-v-12 pvar-location-u-11 s 23 8)
      (cmi::f-subtract-multiply-constant-3-always pvar-location-v-12
                          pvar-location-v-12 -!!-index-2 r 23 8)

      (cm:get-from-north-always -!!-index-2 pvar-location-u-11 32)
      (cmi::f-always slc::stack-index -!!-index-2 23 8)
      (cmi::get-from-north-with-f-subtract-always pvar-location-v-12 pvar-location-u-11 23 8)

      (cm:get-from-south-always -!!-index-2 pvar-location-u-11 32)
      (cmi::float-subtract pvar-location-v-12 slc::stack-index -!!-index-2 23 8)
      (cmi::get-from-south-with-f-subtract-always pvar-location-v-12 -!!-index-2 23 8)
      )
)
```

Obviously the generated assembly code is horrendously complex. If the user had to write this code, the Connection Machine would not be selling today - even if the performance was higher than 3.8 Gflops! The key to the success of Thinking Machines in the last two years has been to produce a compiler that generates such code automatically, and there is where the user interface is most enhanced by the compiler. The development of an optimizing compiler of this quality, addressing *communication* instructions as well as computational instructions is a major achievement of the CM-2 software system. Because of its power, the compiler is essentially the complete user interface to the machine.

3. DEBUGGING TOOLS

The debugging of code is a fundamental user interface issue. On parallel machines, and especially on distributed memory systems, program debugging can be extremely frustrating. Basically one is debugging not one program, but possibly 128 programs. Even if they are all executing the same code, they are not executing the same instructions if the system is MIMD. Furthermore there are synchronization and communications bugs that can make it extremely

difficult to debug anything. For example, one problem that can occur on distributed systems is that intermediate nodes that are required for passing data back for debugging from the node where a suspected bug has developed, may themselves be sick in some form or another. Debugging messages sometimes arrive in a different order than you sent them in, and in any event may well be coming in multiples of 128 (or more). And finally, the overall complexity of the systems can be extremely confusing, particularly when communication data structures involve complex numbering schemes such as Grey codes.

We would like to give an example of a debugging tool that we have developed, and which we have worked with over some time with good experiences. The tool is a parallel distributed-system simulator called PARSIM. One goal of PARSIM was to develop an extremely simple and portable simulator that could be easily instrumented and coupled with visualization.

Portability is achieved by developing a UNIX based tool, where the lowest level communication is implemented through a very simple data transfer capability. The data transfer may be handled using either IP facilities or even just by using the UNIX file system. PARSIM provides library support for Intel Hypercube functionality and also library support for other similar communication capabilities. All of the standard communication protocols are supported including typed messages, broadcasts and global operations. Finally PARSIM is usable from Fortran or C. In fact a user simply links the host and node programs of his application to the PARSIM library.

PARSIM maintains a full trace history of all communication activity. A portable X-11 interface provides a graphical view of all the communication activities so that as the simulation is running one can monitor all communication traffic between nodes. The graphical display represents nodes by colored numbered circles, and messages by arms reaching out from the nodes: a dispatched message is represented by an arm reaching towards the destination, while a receive request is represented by an arm reaching out from the receiver. When a requested message is actually received the corresponding send and receive arms are linked to form a single path indicating a completed transaction. Nodes awaiting message receipt are highlighted and the types of all messages are displayed. In addition to the main display, separate text windows are used to display the output of all node and host processes. Thus the user can watch the communication activity on a global scale while maintaining the ability to follow details on individual processors. The display works effectively on up to 32 nodes, although typically a smaller number suffices to debug most programs. Finally PARSIM provides a history file that records the correct time sequence in which events are occurring. The history file may be viewed later to recheck aspects of a run without the need to rerun the whole program.

PARSIM has turned out to be a key to porting large programs to a whole range of parallel machines, including the Intel iPSC/860. It is much easier to get the programs running in this environment than it is on the Intel. Once applications are running on the simulator, they port to the machine very quickly. As a recent example, with Charbel Farhat of the University of Colorado, we have ported a large (60,000) line finite element program to both the iPSC/860 and the SUPRENUM-1 computers in just several weeks. Thus user interface tools of this type can be extremely helpful.

4. HIGH LEVEL LANGUAGES, EXTENSIONS, LIBRARIES AND GRAPHICS

There has been substantial progress recently in the area of high-level languages for parallel systems. One class of developments has occurred in the area of object-oriented programming models to facilitate parallel programming. An example that I've been involved with was a C++ library for vector and matrix operations, which was implemented on the Intel hypercube, the FPS T-series, the Ametek 2010, the Connection Machine CM-2, and several other systems. Another example is the language DINO developed by R. Schnabel and co-workers at the University of Colorado[4].

There are also some language extensions of several standard languages that are extremely important because they have a better chance of becoming standards. An example here would be the Fortran 90 flavors, for example, Connection Machine CM Fortran. A user can write pure Fortran 90 programs and compile them unchanged on the CM-2, although for best performance it is advisable to insert some compiler directives. This provides the possibility of writing a program for the Connection Machine that might also run on other parallel machines - for example on a CRAY Y-MP. Indeed now that many manufacturers appear to be moving (slowly) towards Fortran 90, there are real possibilities in this direction. Several similar extensions of other languages are now available on multiple systems. One good example would be the Connection Machine language extensions C* of the C language. The C* language is now available on Sequent, Masspar and several other systems.

There are some problems with the high-level language approach. One is the lack of portability. There is increased learning time for users if they have to learn not only the lower-level aspects of systems but also how to deal with new language constructs. Finally there is the danger of the software systems simply becoming too specialized.

A few words are in order about libraries. While there is a tremendous amount of very interesting work going on in the library area, we will not attempt to review it. A very good example is the LAPAK work. Most of that work is going on for shared-memory systems although there are some developments also for distributed memory machines. It is difficult to develop efficient libraries in a way that includes both shared and distributed systems.

For distributed memory systems there is now substantial effort to develop communication libraries that run on multiple systems. One example worth noting is the Suprenum system where they have developed high-level libraries for 2-D and 3-D grid applications[5]. That library really helps the user who has a grid problem to deal with. In fact it allows him to completely dispense with explicit communication calls. He specifies a few communication parameters (topology, grid size etc.) and then handles all interprocess communication through high-level geometrically intuitive operations. The system partitions data sets effectively and calls low-level communication operations to exchange or broadcast data as needed.

One other point to note is that most codes in the real world don't use libraries very heavily, and so one has to be aware that not only is it important to port libraries, but the technology used to design and implement algorithms in the libraries needs to be made available to the scientific community in a way such that other users can adapt those same techniques into their codes.

The graphics area is certainly one of the weakest features of parallel systems. The Connection Machine is really the only system that has tightly coupled graphics capabilities - it supports up to eight hardware frame buffers each running at 40 MB/sec. One disadvantage with the Connection Machine solution is that graphics applications using the frame buffer are not portable. However obviously the ability to at least do high-speed graphics outweighs this disadvantage. In most systems, even if there is a hardware I/O capability that is fast enough, there is a lack of software to support graphics over that I/O channel. Furthermore many systems actually force all graphics to pass through a time-shared front end processor, and an Ethernet connection, ensuring poor performance under almost any conditions.

The best solution is certainly to tightly couple parallel systems to conventional graphics systems. This is a way to avoid getting into graphics systems that are specialized to specific pieces of parallel hardware. Much effort is now underway at several laboratories to develop such high-speed connections between parallel systems and graphics stations.

We will mention an experiment we've done at the Center for Applied Parallel Processing (CAPP) in Boulder where we have connected a Stardent Titan, and a Silicon Graphics IRIS, directly to the back end of a Connection Machine CM-2, allowing data to be communicated between the CM-2 hardware and the graphics system at a very high speed, limited, in fact, only by the back-end speed of the CM-2[6]. The data is postprocessed on the graphics processor using the very high polygon processing rates that are available on those two systems. This was first implemented as a low-level capability based on IP type protocols. However once it was available we realized that it could be used to extend the domain of powerful objected oriented graphics systems, such as the Stardent AVS system, to include objects resident on the CM-2. From this point of view a Stardent user programs the graphics device as if the CM-2 is part of his system.

One of the advantages here is that the Connection Machine can continue with its own numeric processing without waiting for the user to complete visualization. For example the CM-2 can go on to the next time step of a fluid code while you graphically contemplate what has been computed to date.

Another point about this approach (based on a low-level standard protocol) is that it is easy to run the same software over slow connections. In fact we have designed this software so that if the direct connection to the back end is not available, it automatically switches to using Ethernet links. This means you don't have to be in the next room to the CM-2 hardware in order to use the visualization system. Of course, you won't get the same performance from the graphics, but at least the functionality is there.

5. FUTURE SUPERCOMPUTING ENVIRONMENTS: HETEROGENEOUS SYSTEMS

Over the last 20 years we have seen a gradual evolution from scalar sequential hardware to vector processing and more recently to parallel processing. No clear consensus has emerged on an ideal architecture. The trend to vector and parallel processing has been driven by the

computational needs of certain problems, but the resulting systems are then inappropriate for other classes of problems. It is unlikely that in the near term this situation will be resolved, and indeed one can anticipate further generations of even more specialized processor systems appearing. There is a general consensus that a good computing environment would at least provide access to the following resources:

- Scalar processors (e.g. workstations).
- Vector processors.
- Parallel machines: SIMD and/or MIMD
- Visualization systems.
- Mass-storage systems.
- Interfaces to networks.

This leads us to the last topic and what in the long run is probably the most important: heterogeneous systems and heterogeneous environments, and the importance of combining together a spectrum of computing resources.

There is a simple way of avoiding the specialization problem described above. The key is to develop seamless integrated heterogeneous computing environments. What are the requirements for such systems? Obviously high-speed communication is paramount - that means both high bandwidth and low latency. Because different types of machines are present, a seamless environment therefore requires support for data transformations between the different kinds of hardware. Equally important, as I've argued in section 2 from previous experience with single machines, is to try to support wherever possible shared-memory concepts. Ease of use will require load balancing. If there are three Connection Machines on that system, one should be able to load-balance them between the demands of different users. Shared file systems should be supported, and so on. And all of this should be done in the context of portability of the user's code, because the user may not always design his codes on these systems initially. Obviously the adoption of standards is critical.

Such an environment will present to the user all of the resources he might need to avail of for any application: fast scalar, vector and parallel processors, graphics supercomputers, disk farms and interfaces to networks. All of these units would be interconnected by a high bandwidth low latency switch, which would provide transparent access between the systems. System software would present a uniform global view of the integrated resource, provide a global name space or a shared memory, and control load balancing and resource allocation.

The hardware technology is now at hand to allow such systems to be built. Two key ingredients are the recent development of fast switching systems and the development of high-speed connection protocols and hardware implementing these protocols, standardized across a wide range of vendors. We illustrate with two examples.

Recently CMU researchers have designed and built a 100 Mbit/sec switch called NECTAR which supports point to point connections between 32 processors[7]. A Gbit/sec version of NECTAR is in the design stage. Simultaneously, various supercomputer and graphics workstation vendors have begun to develop high speed (800 Mbit/sec) interfaces for their systems.

Combining these approaches we see that at the hardware level it is already possible to begin assembling powerful heterogeneous systems. As usual the really tough problems will be at the software level.

Several groups are working on aspects of the software problem. In our own group at the Center for Parallel Processing in Boulder, Colorado, we have recently developed, as discussed in the previous section, a simple heterogeneous environment consisting of a Connection Machine CM-2 and a Stardent Titan graphics super work-station[6]. The Titan is connected with the CM-2 through the CM-2 back-end, rather than through the much slower front-end interface which is usually used for such connectivity. The object-oriented high-level Stardent AVS visualization system is then made directly available to the CM-2 user, allowing him to access and manipulate graphical objects computed on the CM-2 in real time, while the CM-2 is freed to pursue the next phase of its computation. Essentially this means that to the user, AVS is available *on* the CM-2. Porting AVS directly to the CM-2 would have been a formidable and pointless task. Furthermore the CM-2 is freed to perform the computations that it is best suited for, rather than wasting time performing hidden surface algorithms or polygon rendering. These are precisely the sorts of advantages that can be realized in a heterogeneous system.

Looking to the future, we believe that most of the research issues of heterogeneous computing will have been solved by the late 1990's, and in that time frame we would expect to see evolving heterogeneous systems coming into widespread use wherever a variety of distinct computational resources are present. In the meantime one can expect to see more limited experiments in heterogeneous environments at major research computation laboratories, such as Los Alamos National Laboratory and the NSF Supercomputer Centers.

5.1. An Application for a Heterogeneous System

We will conclude by asking if there are problems that can use such systems? We will answer the question by giving an example which is typical of many real-world problems.

The scientific application is the aeroelastic analysis of an airframe. The computational problem is complicated because there are two distinct but coupled subproblems to be solved. The real aerodynamic design problem, is not just to compute air flow over a plane in the context of a static frame, but to design planes the way they are flown, which is dynamically. As soon as you do that, you get into the coupled problem of the fluid flow and the airframe structure. There is a structural engineering problem, which is represented by the finite element analysis of the fuselage and the wing. There is a fluid dynamics problem, which is the flow of air over the wing surface. Finally there is a real interaction between these two problems. Obviously the lift depends on the wing surface, but correspondingly the fluid flow can cause vibrations in the wing.

One can represent the computation schematically as an two-dimensional (surface) structural problem with $O(N^2)$ degrees of freedom and a three dimensional fluid problem with typically $O(N^3)$ degrees of freedom. Here N measures the spatial resolution of each problem.

Typically the fluid computation can be solved by explicit methods but the finite element problem requires implicit solution techniques. The fluid models require solution time proportional to the number of degrees of freedom. The finite element problems are typically sparse matrix problems and it is hard to do better than $O(N^3)$ on solution time these. Thus we have a computation that is $O(N^3)$ for both the fluid and structure components.

Thus the two computational phases are much the same in overall computational complexity. However, the communication of data, the interaction between the two phases, is related to the surface only and is therefore an $O(N^2)$ data-transfer problem. Therefore provided one is solving a large enough problem so that the $O(N^2)$ communication cost is negligible, one can imagine solving the fluid part on a computer that is most effective for fluid computation and the structural part on a machine that is most effective for structural problems.

We have been studying this problem quite actively at the University of Colorado[8-10] We have found that the fluid problem is best done on a machine like the Connection Machine where one can take advantage of the SIMD architecture and work with grids that are logically rectangular. The structural problem is best done on machines that can handle irregular grids effectively, for example a CRAY Y-MP. Thus ideally we would like to solve the whole problem on a heterogeneous system that includes both CRAY and CM-2 machines. One should also remember that both phases of this computation have heavy visualization requirements. Thus both systems need to be tightly coupled to high-end graphics systems. Furthermore if a record is to be saved of a simulation then high-speed access to a disk farm is mandatory due to the huge volumes of data generated per run. A fully configured heterogeneous environment is therefore essential.

6. CONCLUSION

We conclude by remarking that the winning strategy in supercomputer design for the coming decade is certainly going to be acquisition of a software advantage in parallel systems. Essentially all of the MIMD manufacturers are likely to be using competitive hardware. It is clear from recent experiences -- for example, Evans and Sutherland -- that using anything other than off-the-shelf components is too expensive. It follows immediately that different manufacturers can follow more or less the same upgrade strategies. They will end up with machines with basically similar total bandwidths, reliability and so on. Thus the key to success in this market is going to be to develop systems that have the best software. Key points there will be virtual shared-memory environments and general-purpose computing capabilities.

References

1. O. McBryan and R. Pozo, ''Performance Evaluation of the Myrias SPS-2 Computer,'' CS Dept Technical Report CU-CS-505-90 (submitted to Concurrency: Practice and Experience), University of Colorado, Boulder, 1990.
2. O. McBryan, ''Optimization of Connection Machine Performance,'' *International Journal of High Speed Computing*, vol. 2, no. 1, pp. 23-48, 1990.
3. O. McBryan, ''The Connection Machine: PDE Solution on 65536 Processors,'' *Parallel Computing*, vol. 9, pp. 1-24, North-Holland, 1988.
4. M. Rosing and R. B. Schnabel, ''An Overview of Dino -- A New Language for Numerical Computation on Distributed Memory Multiprocessors,'' in *Parallel Processing for Scientific Computation*, ed. G. Rodrigue, pp. 312-316, SIAM, Philadephia, 1989.
5. K. Solchenbach and U. Trottenberg, ''SUPRENUM - System essentials and grid applications,'' *Parallel Computing*, vol. 7, North Holland, 1988.
6. L. Compagnoni, S. Crivelli, S. Goldhaber, R. Loft, O. McBryan, A. Repenning, and R. Speer, *A Simple Heterogeneous Computing Environment: Interfacing Graphics Workstations to a Connection Machine.,* CS Dept Technical Report, University of Colorado, Boulder, 1991, in preparation.
7. Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., and Steenkiste, P. A., ''The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers,'' in *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 205-216, ACM, Apr 1989.
8. C. Farhat, S. Lanteri, and L. Fezoui, ''Mixed Finite Volume/Finite Element Massively Parallel Computations: Euler Flows, Unstructured Grids, and Upwind Approximations,'' in *Unstructured Massively Parallel Computations*, ed. R. Voigt, MIT Press, 1991.
9. A. Saati, S. Biringen, and C. Farhat, ''Solving Navier-Stokes Equations on a Massively Parallel Processor: Beyond the One Gigaflop Performance,'' *International Journal of Supercomputer Applications*, vol. 4, no. 1, pp. 72-80, 1990.
10. C. Farhat, N. Sobh, and K. C. Park, ''Transient Finite Element Computations on 65,536 Processors: The Connection Machine,'' *International Journal for Numerical Methods in Engineering*, vol. 30, pp. 27-55, 1990.